Sentinel-controlled Input, Selection Mechanisms

This programming assignment uses many of the ideas presented in sections 3 through 6 of the course notes, so you are advised to read those carefully. Read and follow the following program specification carefully. This program is somewhat more complex than Project 2, so don't underestimate it.

You may receive two scores on this project. You will receive a score for runtime testing from the NAG. You may also receive a software engineering score for following the instructions in the Programming Standards section below.

The Program Specification: Party Music

You are putting together a music tape for a party and have arranged a list of songs in the order in which you want to play them. However, you do not know whether all the songs will fit on one side of the cassette tape you have, which will hold 45 minutes on each side. You would like to fit as many songs as possible on one side of the tape, keeping them in the order you've already determined. This may require leaving out one or more of the songs you've selected.

Input file description and sample:

Your program **must** read its input from a file named `music.data` — use of another input file name will result in a score of zero. The first line of the input file contains column labels, which should be ignored. The next several lines of the input file will each contain:

- A positive integer indicating the position of the song in the list. These song numbers will be in sequential order.
- A character indicating the musical category of the song: 'B' for blues, 'C' for country, 'H' for hip-hop, 'P' for pop, and 'R' for rock.
- Two nonnegative integers giving the length of the song in minutes and seconds.

The values will be separated by spaces, and a single newline character will follow the last value. A line containing a single zero will follow the last line of song data.

You may assume that all the input values will be logically correct (no negative or missing values, for instance). Here is an example:

```
Song  Type   Minutes  Seconds
  1     C       5        32
  2     C       7        51
  3     B       6         5
  4     R       5        29
  5     H       9        26
  6     B       4        10
  7     P       7        45
  8     R       5        11
  9     H       1        17
  0
```

Note that you must **also not make any assumptions** about the number of lines of data in the input file. Your program must be written so that it will detect the sentinel value (zero) indicating the end of the input, and stop reading input at that point in the file. Be warned: it is entirely possible the input file may contain additional data following the sentinel line; your program should never read that data.

What to Calculate:

Your program must determine which songs are to be included on the tape. When you read the data for a song, you will add that song to the tape if it will fit in the remaining space (time) available. If the song won't fit, you will reject it and go on to the data for the next song, if there is one.

When you add a song to the tape, you must also calculate the total time used, and the time remaining on the tape. You must also count how many songs in each of the given categories are included on the tape.

Output description and sample:

Your program must write its output data to a file named `tape.list` — use of any other output file name will result in a score of zero. A sample output file produced from the sample input file above is shown below:

```
Programmer:  Marc Bjorklund
Party Music

Song          Song Time           Total Time
Number    Minutes   Seconds    Minutes   Seconds
-----------------------------------------------
   1          5        32          5        32
   2          7        51         13        23
   3          6         5         19        28
   4          5        29         24        57
   5          9        26         34        23
   6          4        10         38        33
   7          7        45      song will not fit
   8          5        11         43        44
   9          1        17      song will not fit
-----------------------------------------------

Time remaining
   Minutes    1
   Seconds   16

Song categories
   Blues      2
   Country    2
   Hip-Hop    1
   Rock       2
```

The first line of your output should identify you by name, as shown. The second line should include the title “Party Music” only. The third line should be blank.

Next your output file will contain a table, with one line of output for each song given in the input file. Each line of the table should list the song number, the length of the song (in minutes and seconds), and the total time used on the tape. If the song will not fit on the tape, the line in the table should list the song number, the length of the song, and the message "song will not fit". The table columns should have labels, exactly as shown. There should be a line of delimiters immediately after the column labels, and another to mark the end of the table. A newline character should follow each line of output (including the last).

After the table, your output file will display the amount of time left on the tape (in minutes and seconds), using the format and labels shown below. After that, your output file will display the number of songs on the tape that are in each of the

given categories. A category should be listed only if at least one song from that category is included on the tape. Again, you must use the format and labels shown below. The categories must also be shown in alphabetical order.

You are not required to use the exact horizontal spacing shown in the example above, but your output must satisfy the following requirements:

- You must use the specified header and column labels, and include your name in the first line as shown.
- You must arrange your output in neatly aligned columns, with a label identifying the contents of each column. Use spaces, not tabs to align your output. Note that while the NAG doesn't deduct points for horizontal alignment, the TA who grades your source code for programming standards may do so.
- You must use the same ordering of the columns as shown here.
- You must print a newline at the end of each line.

Programming Standards:

You'll be expected to observe good programming/documentation standards. All the discussions in class about formatting, structure, and commenting your code will be enforced. A copy of *Elements of Programming Style* is included with the course notes — if you don't have a copy I strongly suggest you read the on-line edition (available from the course web page). Some specifics:

- You must include header comments specifying the compiler and operating system used and the date completed.
- Your header comment must describe what your program does; don't just plagiarize language from this spec.
- You must include a comment explaining the purpose of every variable or named constant you use in your program.
- You must use meaningful identifier names that suggest the meaning or purpose of the constant, variable, function, etc.
- Precede every major block of your code with a comment explaining its purpose.
- You must use indentation and blank lines to make control structures like loops and if-else statements more readable.
- Use named constants instead of variables where appropriate.

You are also required to use a `switch` statement in this program — where is up to you, but I think there's a natural place to use one.

Your submission that receives the highest score may be graded for adherence to these requirements, whether it is your last submission or not. If two or more of your submissions are tied for highest, the earliest of those will be graded. Therefore: implement and comment your C++ source code with these requirements in mind from the beginning rather than planning to clean up and add comments later.

Incremental Development:

You'll find that it's easier and faster to produce a working program by practicing incremental development. In other words, don't try to solve the entire problem at once. First, develop your design. When the time comes to implement your design, do it piece by piece. Here's a suggested implementation strategy for this project:

- First, write the code necessary to read the entire input file. To test your work, include code to write what you're reading (and nothing else) to the output file. There's not much point in worrying about processing the data further until you know you're reading it correctly, and stopping at the right point.
- Second, add the code to decide whether to include the current song on the tape, and keep track of how much time has been used and how much time is left.
- Third, add the code to count the number of included songs in each category and print that information.
- Fourth, clean up your output format and be sure it matches the specification above.

Now you have a substantially complete program. At this point, you should clean up your code, eliminating any unnecessary instructions and fine-tuning the documentation you already wrote. Check your implementation and output again to be sure that you've followed all the specifications given for this project, especially those in the Programming Standards section above. At this point, you're ready to test your program on all the posted input/output examples before submitting your solution to the NAG.

Hints:

This program requires that you know how to manage a sentinel-controlled input loop — examples of this are shown in the course notes on slides 6.6 and 6.9.

Be careful to identify and correctly handle the boundary cases in this problem.

Testing:

Obviously, you should be certain that your program produces the output given above when you use the given input file. However, verifying that your program produces correct results on a single test case does not constitute a satisfactory testing regimen.

At minimum, you should test your program on **all** the posted input/output examples given along with this specification. The same program that will be used to test your solution generated those input/output examples. You could make up and try additional input files as well; of course, you'll have to determine by hand what the correct output would be.

Pledge:

Each of your submissions to the NAG must be pledged to conform to the Honor Code requirements for this course. Specifically, you **must** include the following pledge statement in the header comment for your program:

```
//    On my honor:
//
//    - I have not discussed the C++ language code in my program with
//      anyone other than my instructor or the teaching assistants
//      assigned to this course.
//
//    - I have not used C++ language code obtained from another student,
//      or any other unauthorized source, either modified or unmodified.
//
//    - If any C++ language code or documentation used in my program
//      was obtained from another source, such as a text book or course
//      notes, that has been clearly noted with a proper citation in
//      the comments of my program.
//
//    - I have not designed this program in such a way as to defeat or
//      interfere with the normal operation of the Automated Grader.
```

Failure to include this pledge in a submission is a violation of the Honor Code.